# Efficient Asynchronous Interrupt Handling in a Full-System Instruction Set Simulator

# Efficient Asynchronous Interrupt Handling in a Full-System Instruction Set Simulator

Tom Spink Harry Wagstaff Björn Franke

Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, UK

t.spink@sms.ed.ac.uk, h.wagstaff@sms.ed.ac.uk, bfranke@inf.ed.ac.uk

## Abstract

Instruction set simulators (ISS) have many uses in embedded software and hardware development and are typically based on dynamic binary translation (DBT), where frequently executed regions of guest instructions are compiled into host instructions using a just-in-time (JIT) compiler. Full-system simulation, which necessitates handling of asynchronous interrupts from e.g. timers and I/O devices, complicates matters as control flow is interrupted unpredictably and diverted from the current region of code. In this paper we present a novel scheme for handling of asynchronous interrupts, which integrates seamlessly into a region-based dynamic binary translator. We first show that our scheme is correct, i.e. interrupt handling is not deferred indefinitely, even in the presence of code regions comprising control flow loops. We demonstrate that our new interrupt handling scheme is efficient as we minimise the number of inserted checks. Interrupt handlers are also presented to the JIT compiler and compiled to native code, further enhancing the performance of our system. We have evaluated our scheme in an ARM simulator using a region-based JIT compilation strategy. We demonstrate that our solution reduces the number of dynamic interrupt checks by 73%, reduces interrupt service latency by 26% and improves throughput of an I/O bound workload by 7%, over traditional per-block schemes.

***Categories and Subject Descriptors*** D.4.8 [*Operating Systems*]: Performance—Simulation

***General Terms*** Design, experimentation, measurement, performance

***Keywords*** Full system simulation, Dynamic binary translation, Region-based just-in-time compilation, Asynchronous interrupt handling

## 1. Introduction

Instruction set simulators (ISS) are indispensable tools for software and hardware developers alike. In fact, in the embedded systems domain, ISS are routinely used during software development stages for functional and performance testing whilst hardware designers rely on their fast turn-around times for prototyping new architectures or architectural extensions.

Full-system ISS have extended capabilities that go beyond the execution of a stream of user mode instructions in standard ISS. Such full-system ISS need to support additional features including a memory management unit, kernel mode instructions, interrupt handling, and device emulation, which are required for the simulation of a complete system capable of hosting an operating system (OS).

Efficiency of interrupt handling is of particular importance as interrupts need to be processed frequently (e.g. the frequency of the Linux kernel timer interrupt which triggers the process scheduler is typically set between 100-1000Hz) and require fast response. Unfortunately, efficient interrupt handling is at odds with dynamic binary translation (DBT), which is the underlying technology for building high-performance ISS. In a DBT-based ISS control flow is profiled and regions of frequently executed guest instructions are identified and translated to host instructions using a just-in-time (JIT) compiler [25]. This generated native code executes much faster than an interpreter can execute guest instructions, thus providing higher simulation performance. Interrupts, however, interfere unpredictably with the "natural" control flow of an application and divert it away from the current region of code to another. To capture this behaviour additional checks need to be inserted into the generated code, which initiate interrupt handling if a pending interrupt request is signalled. These additional checks are costly to perform and can inhibit aggressive region-based optimisations resulting in a reduction of simulation performance by more than an order of magnitude, if inserted naïvely e.g. after each guest instruction.

For *asynchronous* interrupts, i.e. externally triggered interrupts unrelated to the currently executing instruction and the current state of the processor, handling can be deferred by a small period of time until a more "convenient" moment. For example, an OS might mask certain interrupts in critical sections and only process pending interrupts after leaving such a section. We exploit this trait in our ISS and insert fewer interrupt checks, thus reducing their performance impact.

The central questions we are trying to answer in this paper are: What is the *minimum* number of interrupt checks that need to be inserted and *where* to insert them?

We show that inserting interrupt checks at the beginning and end of linear execution traces, e.g. generated by trace based JIT compilers, does not work with region based JIT compilation schemes, which offer higher simulation rates than their trace based counterparts.

In this paper we develop a new scheme for inserting asynchronous interrupts checks in an ISS using a region based DBT. We dynamically identify control flow loops and insert appropriate interrupt checks within each control flow cycle to maintain correctness. This is important for loops which depend on interrupt

handling for their termination. Whilst inserting a strictly minimal number of interrupt checks is an NP-hard problem [8], we utilise an existing approximation algorithm suitable for use in a performance-critical JIT environment, which for most practical cases computes an almost optimal solution. We show how our region profiling interacts with interrupt handling such that nested interrupts can be processed and interrupt handlers themselves presented to the JIT compiler, enabling further performance improvements. We demonstrate the efficiency of our interrupt handling scheme using our region based DBT ISS targeting the ARM instruction set, reducing the number of dynamic interrupt checks by 73%, reducing interrupt service latency by 26% and improving the throughput of an I/O-intensive workload by 7%.

### 1.1 Motivating Example

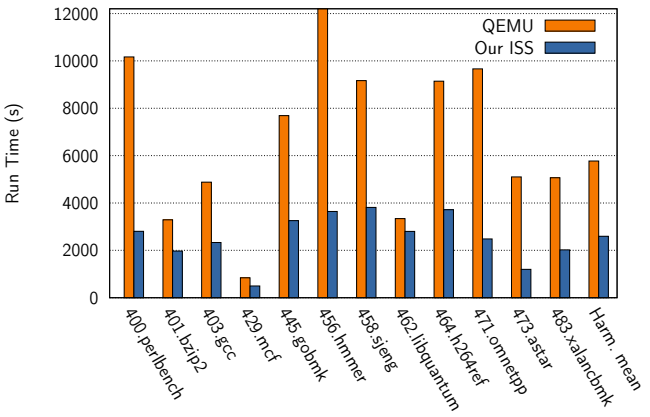



Figure 1: In user-mode simulation our region-based ISS clearly outperforms QEMU. Aggressive region-based optimisations substantially reduce absolute run times (in seconds, lower is better) of the SPEC CPU2006 benchmarks. Region-based DBT, however, presents a challenge to *full-system simulation*, where interrupt checks need to be performed. We present a methodology, which is (a) correct and (b) retains the performance advantage of region-based DBT in a full-system simulation context.

A *basic-block*-based DBT is one which translates *guest* basic-blocks one-at-a-time into corresponding *host* code (see Figure 2a). These blocks are treated as independent units, and control-flow is performed by jumping to existing code in a cache, or causing an on-demand translation if the code has not yet been seen. Extending a basic-block based DBT to consider multiple blocks along a path leads to a *trace*-based DBT, which allows for optimisations to cross basic-block boundaries, as the trace is considered its own unit (see Figure 2b). Whilst trace-based DBTs consider linear control flow only, a *region*-based DBT can exploit control-flow (such as loops) within a particular region of code (see Figures 2c and 2d).

User-mode simulation of applications require only that a target binary is emulated on the host, and do not usually require interrupts or device emulation – a simple OS emulation layer is enough to execute most user binaries. Notable exceptions are applications that require or depend on *asynchronous* Unix signals, but the majority of benchmarks (and certainly those present in the SPEC CPU2006 suite [13]) do not depend on this behaviour and so are ill-suited to testing this important requirement of full-system simulation. Figure 1 shows that in user-mode simulation, where interrupt checks can be ignored, our simulator performs on average 2.23*x* faster than QEMU [4], also in user-mode configuration.

The increase in simulation performance seen in Figure 1 is due to our region-based approach (implementing techniques such as optimised end-of-block handling [25]), and down to the aggressive optimisations that we can perform within a region of code.

```
1  void main()
2  {
3      // Wait for some hardware signal
4      while (received_irq == 0) usleep(10);
5
6      //Now do some computation in a loop
7      for (int i = 0; i < 10; ++i) {
8          output[i] = inputa[i] * inputb[i]
9          if((i & 1) == 0) {
10             output[i] += inputc[i];
11         } else {
12             output[i] -= inputc[i];
13         }
14     }
15 }
```

Figure 3: Example code requiring interrupt checking. Termination of the **while** loop in `main` is dependent on an interrupt, hence we must insert an interrupt check inside this loop. Termination of the **for** loop is not dependent on an interrupt, but delaying interrupt checking until after the loop may introduce an unacceptably large interrupt latency. We want to insert as few as possible interrupt checks in this loop such that we (a) check on every iteration, and (b) do not perform more checks than necessary for performance reasons.

However, it is possible to hinder these optimisations by inserting additional side exits into a region. Such an additional side exit may be an interrupt check, where a flag is tested to determine if an interrupt is pending, and then exit the region so that the simulator may process the pending interrupt. It is therefore desirable to reduce the amount of these side exits, so that our aggressive region optimisations can continue to be effective in full-system simulation, where interrupt checks are mandatory.

As of later versions, QEMU's approach to interrupt checking is to insert a check at the head of every translation block, resulting in a check before a guest basic-block is executed. If an event occurs which requires QEMU to leave native code, a flag is set and the native code will exit to the main execution loop where the event is processed. Our approach is similar, in that we maintain a flag to indicate if an *asynchronous action* is pending, and we insert a check into a block to determine if the flag is non-zero. If this condition is met, we will exit native code.

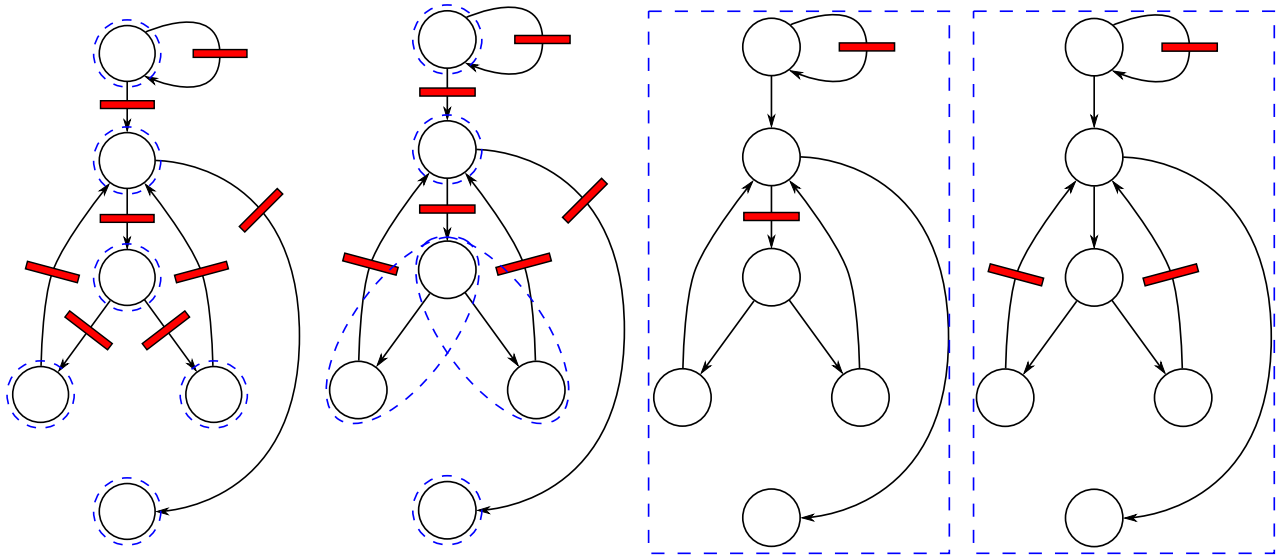
The difference arises when we consider the different approaches taken to native code generation between our ISS and QEMU.

#### 1.1.1 State-of-the-Art QEMU

Since QEMU does not apply any form of *inter*-block optimisation, the problem does not exist that an extra check can affect optimisations – only an easily predicted branch-not-taken penalty will occur. However, since we are considering a region of basic blocks as a unit, and wish to optimise across these basic blocks, the insertion of interrupt checks can inhibit our ability to perform certain optimisations that we could otherwise make in the absence of these checks, and significantly increase the amount of code our JIT compiler must translate.

#### 1.1.2 Correct Handling of Interrupt Dependent Behaviour

Considering the code shown in Figure 3, it can be seen to contain a loop that waits for an interrupt handler to run, followed by a loop-based computation on three input arrays. In order for this program to be simulated correctly, it must be able to receive and process



(a) Block based DBT. Interrupt checks inserted after each basic block, resulting in poor performance due to excessive number of checks.

(b) Trace based DBT. Checks inserted at the start and end of each linear trace. Control flow flattened, prohibiting loop optimisations.

(c) Region DBT. Optimal insertion of interrupt checks. Minimal number, but requires solving an NP-hard problem at JIT compilation time.

(d) Region DBT. Approximation algorithm results in higher number of checks, but is close to optimal and fast enough for JIT compilation.

Figure 2: Interrupt checks, represented by bars on control flow edges, inserted by various interrupt check policies in the control flow graph representing the example program from Figure 3. Translation units (basic blocks, linear traces, regions) are highlighted.

asynchronous interrupts during the initial **`while`** loop. The computation in and termination of the **`for`** loop is not dependent on an interrupt, but we must nonetheless insert an interrupt check to avoid an unacceptably large latency, should an interrupt be signalled. The control flow graph for this code can be seen in Figure 2.

This kind of behaviour is present in full-system simulation, most prominently in operating system kernels, which may wait for an external device to indicate that a buffer is full and ready for processing.

Checking for interrupts in an interpreted-only, functional (i.e., non-cycle-accurate) simulator is straightforward – we can simply check at the end of each interpreted instruction or basic block, or alternatively check after a given number of instructions have been executed. These all produce correct behaviour, and impose varying latencies on servicing the interrupts depending on the scheme we use. For a JIT based DBT selection of the interrupt checking strategy is significantly more challenging, though.

### 1.2 Contributions

In this paper we make the following contributions:

1. We devise a new scheme for the optimised handling of asynchronous interrupts in the context of a region based DBT ISS,
2. we show that our algorithm for inserting interrupt checks is efficient and suitable for JIT processing and does not introduce unbounded interrupt response times, and
3. we demonstrate that our scheme improves simulation performance and I/O throughput on full-system simulation of Linux targeting the ARM instruction set.

### 1.3 Overview

The remainder of this paper is structured as follows. In Section 2 we briefly introduce the problem of interrupt check placement present in a DBT. This is followed by an overview of the various schemes available, and a description of our new interrupt handling scheme in Section 3. In Section 4 we present our empirical evaluation results, before we discuss related work in Section 5. Finally, we summarise and conclude in Section 6.

## 2. DBT Granularity and the Problem of Inserting Interrupt Checks

When implementing interrupt checking in a DBT system, we have many more options for inserting interrupt checks. One of the biggest details in a DBT system (and thus one of the biggest factors in how interrupts are addressed) is whether it translates on a basic block basis, a trace basis, or a region basis (see Figure 2).

Basic block based DBTs must check for interrupts at least once per basic block. This is shown in Figure 2a. Since each block is permitted to be entered from any predecessor, then if we did not perform an interrupt check at the end of a block we may get stuck in a loop waiting for an interrupt which is never detected. As control-flow between basic-blocks in this kind of DBT is relatively straightforward (usually a map lookup from virtual address to translated code), returning from interrupt handlers is also straightforward as the next instruction to be executed will be the head of a basic block and can be looked up from the mapping.

Trace based DBTs have slightly more flexibility in that we can either check for interrupts at the end of each block (within a particular trace), or we can check at the start of each trace. This is illustrated in 2b. Control-flow within a trace is linear, but there may be multiple exit points and so checking at the trace head ensures that if we exit a trace early, we can check for interrupts in the head of the next trace. Checking more frequently may reduce interrupt latency but will impact performance. Checking less frequently may result in the same problem as in the basic block case, where we never detect an interrupt necessary for the simulated program to proceed.

Although the strategies discussed so far work effectively for block and trace based DBT systems, they are inadequate for region based DBTs, which take advantage of dynamically-extracted control flow information to optimise the generated code across basic block boundaries, and to apply certain loop optimisations to a region of code (Figures 2c and 2d). An optimisation phase may even split or merge guest program basic blocks during a transformation pass, which will produce highly optimised and correct behaviour, but the representation of the original basic block will be lost.

Unlike in basic block or trace based DBT systems, the generated translations are able to contain looping control flow, which means certain care must be taken to ensure interrupts are serviced in a timely manner. A naïve DBT system may decide to insert interrupt checks at the end of each translated basic block. However, this negates many of the benefits of a region based DBT as each interrupt check may result in an exit from translated code, making optimisations which span loops and basic blocks much less effective. Making interrupt checks on entry to or exit from a region (as in tracing DBT systems) will also cause incorrect behaviour, as interrupt dependent loops may be encountered within a region, as shown in the example above.

Instead, we should analyse the control flow graph of the region to identify the minimum set of blocks that must contain interrupt checks, while still ensuring correct behaviour. In this case, we must ensure that we have at least one interrupt check in at least one *unconditional* basic block of each loop in the CFG. If we fail to insert an interrupt check into a very long running loop, we may postpone an interrupt for an unacceptable length of time (potentially indefinitely). Furthermore, if we fail to insert an interrupt check into a loop that has behaviour which depends on an interrupt being serviced, then our DBT will behave incorrectly.

An algorithm for computing the minimum set of blocks which must contain interrupt checks can be based on computing the *minimum feedback edge set* [8, 15] of the control flow graph. This identifies the minimum set of edges in the CFG which, when cut, remove all cycles from the CFG. By inserting interrupt checks into the root blocks (*source nodes*) of these edges, we can ensure we do the minimum number of interrupt checks necessary, thus ensuring correct behaviour while maintaining good performance. In our example, this would give us the interrupt checks seen in Figure 2c.

However, in order to ensure good performance in our DBT, we must also ensure that we have good 'warm-up' time. That is to say, we must balance the performance of generated code against how quickly that code can be produced. Computing the exact feedback arc set of a graph is expensive (the problem is NP-hard [8]), whereas computing an approximation is much faster [9], and is unlikely to result in a significant degradation of performance in generated code versus computing the exact feedback arc set. An approximation of the feedback arc set algorithm on our example might give us the interrupt checks shown in Figure 2d, but of course there are many other possibilities.

```
1  define ApplyChecks(WorkUnit):
2    NextIndex := 0
3    do:
4      RMCount := 0
5
6      foreach Block in WorkUnit.Blocks:
7        if Block.HasSelfLoop:
8          Block.HasInterruptCheck := True
9        else if not Block.HasInterruptCheck:
10         call StrongConnect(WorkUnit, Block)
11   while RMCount != 0
12
13 define StrongConnect(WorkUnit, StartBlock):
14   StartBlock.Index := NextIndex
15   StartBlock.LowLink := NextIndex
16   StartBlock.Seen := True
17   NextIndex++
18
19   BlockStack.Push(StartBlock)
20   StartBlock.OnBlockStack := True
21
22   foreach Successor in StartBlock.SuccessorBlocks:
23     if Successor.HasInterruptCheck:
24       continue
25
26     if not Successor.Seen:
27       StrongConnect(WorkUnit, Successor)
28       StartBlock.LowLink :=
29         min(StartBlock.LowLink, Successor.LowLink)
30     else if Successor.OnBlockStack:
31       StartBlock.LowLink :=
32         min(StartBlock.LowLink, Successor.Index)
33
34   if StartBlock.LowLink == StartBlock.Index:
35     Count := 0
36     do:
37       StackedBlock := BlockStack.Pop()
38       StackedBlock.OnBlockStack := False
39       Count++
40     while StackedBlock != StartBlock
41
42     if Count > 1:
43       StartBlock.HasInterruptCheck := True
44       RMCount++
```

Figure 4: Optimised interrupt check placement algorithm for arbitrary code regions, based on Tarjan's [26] algorithm. The algorithm implements the suggestion of maintaining a flag for each node to determine if it exists on the block stack in constant time, and a test to handle blocks which loop to themselves

## 3. Region-based Interrupt Checking

### 3.1 Overview

Our ISS is a region-based DBT that exploits optimisation opportunities within hot regions of code to maximise simulation throughput. Taking collected profiling information, the system applies optimisations across basic-blocks within a particular region to generate highly efficient host machine code that can accurately emulate a target instruction set. There is not necessarily a one-to-one mapping between basic-blocks in target machine code and the code generated by our JIT compiler. The optimisation phases may decide to merge or split basic-blocks arbitrarily and this is a perfectly valid operation, provided the behaviour of the binary being translated is honoured. The profiling information identifies which basic-blocks may be local to a region, i.e. not accessed from outside the region, and so the ability to jump arbitrarily to these blocks is removed. This important distinction allows *region local blocks* to be arbitrarily transformed by the optimisation phase, allowing for optimisations to be performed across these blocks within a region. If we allowed entering (and exiting) the region by any basic-block, then we would have to make certain guarantees about the simulated CPU state (for example, modified register values are written back to the state structure), which would introduce unnecessary overheads to the generated native code.

Execution in our simulator begins by using an interpreter, collecting region information as we progress. Such information includes the heads and lengths of basic blocks, as well as all detected edges between basic blocks, which may be direct branches, predicated branches, or indirect (computed) branches.

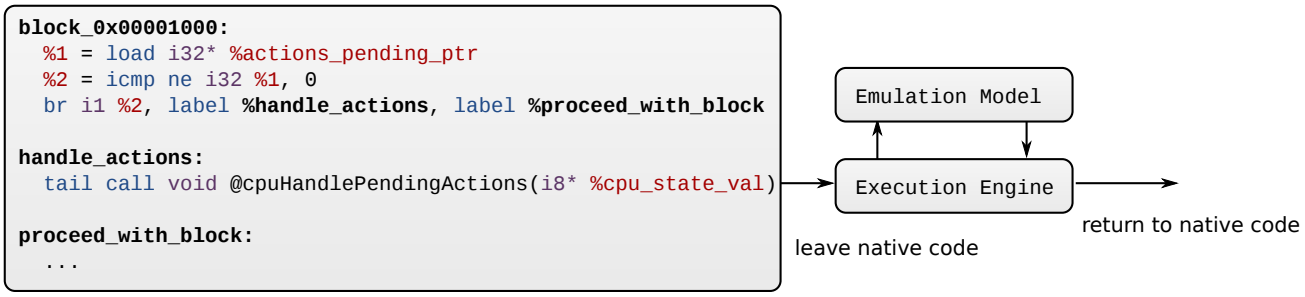



Figure 5: LLVM IR emitted for interrupt checking at the head of a block determined to be an interrupt check block. If the CPU state structure indicates that an action is pending, control leaves native code via a tail-call back to the execution engine, where the pending action is handled.

Once we identify that a region contains *hot* basic blocks (i.e., some blocks in that region have been executed more than $N$ times, where $N$ is a configurable threshold), it is dispatched to an asynchronous JIT compilation task farm. An LLVM [16] bytecode translation is then produced for the region, and standard optimisations are applied, before finally machine code is emitted for the region. During the profiling phase, certain blocks are identified as *region entry points*. These blocks are used to transition from interpreted code to translated code, and serve to enable further optimisations to take place within a region by limiting the number of region entry points only to those which have been discovered. Since we may not necessarily have encountered all code or control flow within a region before we translate it, we may need to return to the interpreter in order to execute untranslated code. In this case, this untranslated code may become 'hot' and make the region eligible for retranslation.

Asynchronous interrupts are a source of adverse control-flow and can significantly degrade collected profiling information by introducing spurious edges from profiled basic-blocks. To account for this, we maintain an *interrupt stack*, which allows for control-flow to be profiled at the currently executing *interrupt level*. By default, we begin execution in a special *no interrupt* level and collect profiling information as execution progresses. When an interrupt check indicates an interrupt is pending, the interrupt level is pushed to the *interrupt stack* and execution continues in the interrupt handler with the profiler now collecting information in the new level. Once the interrupt handler completes (possibly returning to user-code), the interrupt level is popped from the stack and execution continues from where it left off, with profiling information from the point of interrupt maintained. A stack is used to accommodate nested interrupts. Figure 6 shows how the region forming process proceeds in the presence of interrupts. Rather than superfluous edges being formed between block $A$ and block $A'$, and $D'$ and $B$, control flow is discovered as it exists in the original executable.

### 3.2 Interrupt Check Placement Schemes

We have implemented three interrupt check placement schemes in our ISS. Whilst the most accurate algorithm for computing the feedback arc set of the region graph could be used to select basic blocks in which to emit interrupt checks, we instead use an approximate algorithm based on *Tarjan's Strongly Connected Components (SCC)* [26] algorithm as described in Figure 4. The use of the approximation ensures that our ISS retains its fast warm-up time, by reducing the latency introduced in employing this analysis phase.

We always check for interrupts after a basic-block has been executed by the interpreter (regardless of the scheme in use) – our schemes apply to how interrupt checks are inserted by the JIT compiler as it is the performance of generated native code that we are interested in maintaining.

The placement schemes we implement are detailed below:

1. **Full:** An interrupt check is inserted before every basic-block. This is identical to how QEMU performs interrupt checking.
2. **Backwards:** An interrupt check is inserted before every basic-block that is the target of a *backwards* branch.
3. **Optimised:** An interrupt check is inserted before the basic-blocks selected by the algorithm described in Figure 4.

During the compilation phase, a *compilation work unit* (containing *guest* basic-blocks and their control-flow information) is subjected to analysis by the selected interrupt checking scheme, which determines which blocks should contain interrupt checks. Once those blocks are identified, interrupt checks are inserted where necessary by the translator, as each block is translated.

Tarjan's SCC algorithm requires a minor modification to work with our ISS. In particular, we must detect self-loops (a basic-block with itself as a successor) which the algorithm proper does not, and we implement the suggestion for testing whether a node is on the stack in constant time by maintaining an `OnBlockStack` flag for each node. Otherwise, the algorithm remains unmodified.

### 3.3 Taking an Interrupt

As we are dealing with asynchronous interrupts, an interrupt may be asserted by any simulated component, at any time and on any host machine thread. We use the concept of "pending actions" to indicate the presence of an action that must interrupt normal execution, and we use a bitfield in the CPU state structure to indicate what type of action may be pending. It is this bitfield that is queried when determining whether or not an interrupt is pending, and the native code that we emit for interrupt checking simply tests the bitfield. If the value is determined to be non-zero, then we know that an asynchronous action is pending, and that we must leave native code to service it.

A pending action may be an interrupt, a Unix signal or a special internal signal such as "abort" or "dump state". This allows a guest program, for example, to register signal handlers and have a host signal propagated through. Alternatively, if a full operating system is being simulated, during the OS initialisation phase an *interrupt vector table* (IVT) will have been initialised with locations to branch to when a particular interrupt is pending. When an emulated platform device asserts an interrupt, control-flow will branch via this IVT to the correct location in the guest OS.

Figure 5 shows that when an interrupt check block is executed, and the *pending actions* bitfield is non-zero, control returns from native code via a tail-call back into the execution engine. The

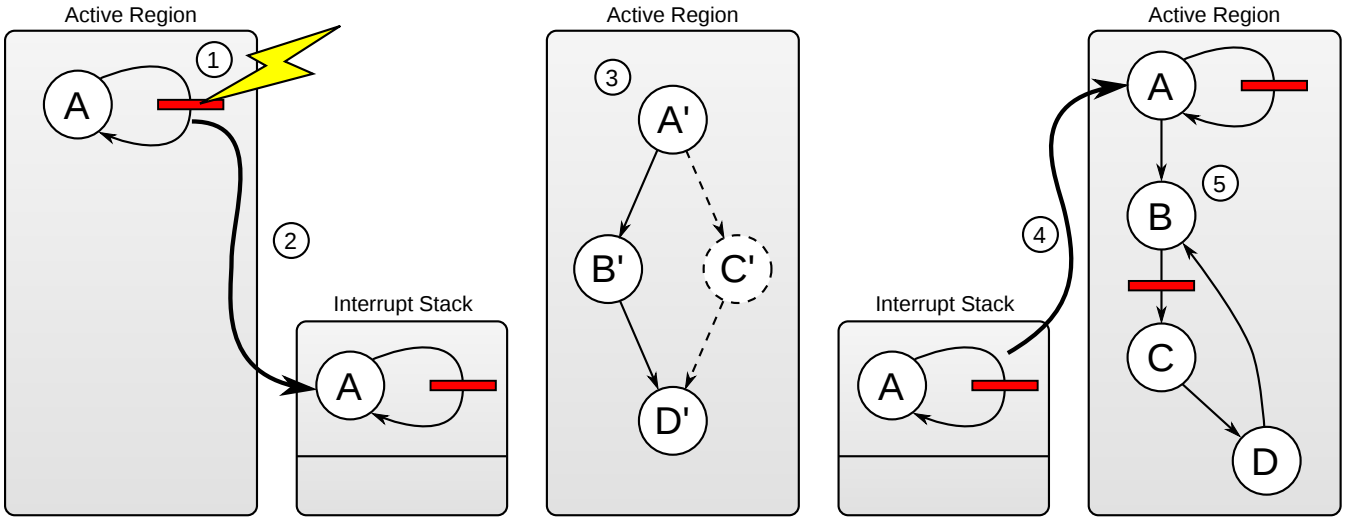



Figure 6: Flow of region forming when an interrupt occurs. At (1), while executing the IRQ-detection loop (and before any other code has been discovered), we detect an interrupt. At (2), the current region state is pushed onto the *interrupt stack*. At (3), we execute the interrupt handler, treating it as a totally distinct region to the original region. At (4), we return to normal execution and pop the previous region state from the stack. At (5), we exit the tight loop and continue forming the original region. Crucially, no superfluous edges which link the 'normal' region to the 'interrupt' region have been created.

execution engine then invokes the necessary routines to service the pending action. Since the result of handling the action may result in adverse control-flow, (i.e. a change to the PC) we cannot return to native code from where we exited and instead must continue execution via the normal execution engine path. This may result in returning to native code, but if the interrupt service routines (ISR) have not yet been compiled, then execution will proceed through the interpreter (potentially marking the ISR as hot).

## 4. Experimental Evaluation

### 4.1 Experimental Methodology

We perform measurements on two different types of workloads to evaluate the impact of our approach. One workload is an interrupt-heavy workload in the form of an I/O benchmark, using the standard Linux I/O benchmarking tool `hdparm` [20], another is a compute-heavy workload using the SPEC CPU2006 integer benchmarks. This comparison will evaluate the impact of our optimisations on workloads that require low-latency interrupt servicing, and those that do not rely on interrupts and hence are not sensitive to interrupt latency.

### 4.2 Experimental Setup

All of our workloads are executed inside an ARM Linux 3.17.0 *guest* operating system (with an Arch Linux ARM user-space), running inside our ISS in full-system mode. The host machine for simulation is described in Table 1a, and the configuration of our ISS is described in Table 1b.

***State-of-the-art*** We also compare ourselves to the state-of-the-art DBT QEMU version 2.1.50. This comparison is to indicate that our region-based approach to compilation can give us significant performance improvements, even with the added complexity of inserting interrupt checks.

#### 4.2.1 Platform Configuration

We are using a *vanilla* (unmodified) Linux 3.17.0 kernel as the simulator's guest operating system to host our experiments. It is configured for an *ARM Versatile Application Baseboard*, and contains no extra configuration or modifications other than enabling the VirtIO block device module. This kernel boots unmodified on both our ISS and QEMU.

The ARM Versatile Application Baseboard includes a single ARM926 CPU, as well as many external devices such as timers and I/O modules. We support many of these devices, excluding those which are irrelevant to our experiments (such as the FPGA), or for which no documentation is publicly available. The platform as specified includes only 128MB of RAM, which is not enough to run the SPEC benchmark suite. For this reason we modify the platform to include additional memory - this modification is made in both our ISS, and in QEMU.

We have implemented the VirtIO specification in our ISS, as detailed in [23] in order to provide a block device implementation to the guest Linux operating system. This block device contains the root filesystem for booting, and is also used as the target of the I/O benchmark for testing.

### 4.3 Main Results for I/O-bound Workloads

For our I/O benchmarking, we are not interested in testing the underlying storage device, but simply wish to stress our interrupt system. Therefore, the `hdparm` benchmark is suitable for these interrupt tests as our I/O device is implemented as a VirtIO [23] block device which uses interrupts to convey I/O completion information back to the guest. We can measure the performance of our system, by measuring the I/O throughput of our system, as I/O throughput will correspond directly to the rate at which we can service interrupts. We do not need to test different I/O access patterns (such as sequential, random, etc), as this will not have any effect on the interrupt system.

In our ISS, when a simulated device raises an interrupt, it causes an IRQ line to become asserted on the CPU, which in turn causes an asynchronous action to become pending. This asynchronous action will eventually cause the execution of the simulated CPU to branch to an interrupt service routine (ISR). We define *interrupt latency* to be time it takes for a simulated device to raise an interrupt and for the CPU to branch to and begin executing the ISR.

| Vendor & Model | Dell™PowerEdge™ R610 |
|---|---|
| Processor Architecture | x86-64 |
| Processor Model | 2× Intel©Xeon™ X5660 |
| Number of cores | 2 × 6 |
| Clock/FSB Frequency | 2.80/1.33 GHz |
| L1-Cache | 2 × 6× 32K Instruction/Data |
| L2-Cache | 2 × 6× 256K |
| L3-Cache | 2× 12 MB |
| Memory | 36 GB across 6 channels |
| Operating System | Linux version 2.6.32 (x86-64) |

(a) ISS Host Configuration.

| ISS Parameter | Setting |
|---|---|
| Guest architecture | ARMv5T |
| Guest operating system | Linux 3.17.0 |
| Host architecture | x86-64 |
| Translation/Execution Model | Asynch. Mixed-Mode |
| Tracing Scheme | Region-based [6, 25] |
| Tracing Interval | 30000 blocks |
| JIT compiler | LLVM 3.4 |
| No. JIT Compilation Threads | 10 |
| JIT Optimisation | `-O3` & Part. Eval. [27] |
| JIT Threshold | Adaptive [6] |

(b) ISS System Configuration.

Table 1: Experimental setup: Host system parameters and configuration of the ISS system.

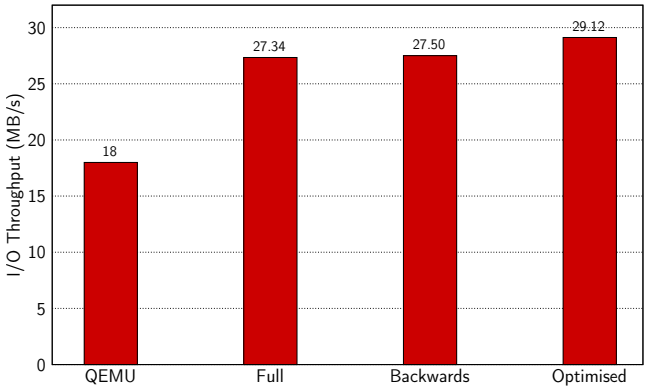



Figure 7: Absolute I/O throughput in MB/s measured with the `hdparm` benchmark – higher is better. We show that in all cases, we have a higher I/O throughput than QEMU, and we improve over our baseline *full* scheme by 7%.

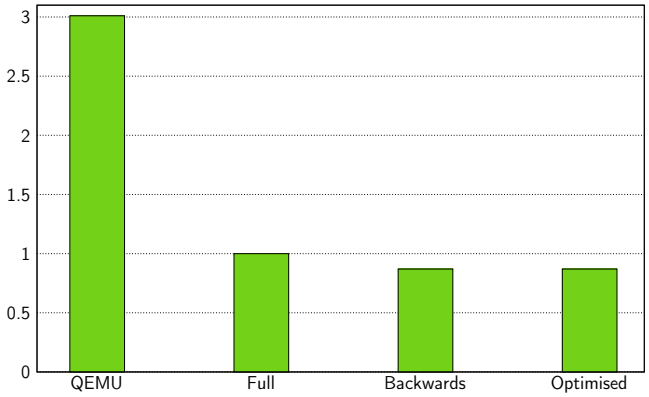



Figure 8: Relative reduction in wall-clock runtime of the SPEC CPU2006 integer benchmark against the **full** baseline – lower is better. We show that in all cases, we are faster than QEMU, and improve our simulation speed by 13% with the optimised placement scheme.

Figure 7 shows a 61% improvement in I/O throughput on the `hdparm` benchmark over QEMU, and a 7% relative improvement when using the optimised placement scheme versus the backwards and full checking schemes.

### 4.4 Main Results for CPU-bound Workloads

As we are simulating a complete operating system, it is not possible to remove all interrupt checks – even when no interrupts are raised for some time – and as such CPU-bound workloads will incur a small performance penalty due to occasional interrupt checking. Therefore, we consider the impact our schemes have on the runtime of a CPU-bound workload. Figure 8 shows that we experience a **13%** reduction in the runtime of the SPEC CPU2006 integer benchmarks when employing the more optimal placement algorithms. This can be attributed to the higher quality of native code that we generate as a result of inserting fewer interrupt checks.

The SPEC CPU2006 integer benchmark is widely used and considered to be representative of a broad spectrum of application domains. We have used it together with its *reference* data sets. The benchmarks have been compiled using the GCC 4.6.0 C/C++ cross-compilers, targeting the ARM architecture (without hardware floating-point support) and with `-O2` optimisation settings.

#### 4.4.1 Comparison to QEMU

An important metric for our ISS is to continue yielding a performance improvement for our CPU-bound workloads over the state-of-the-art QEMU. Figure 8 shows that against our baseline scheme (*full* placement), QEMU is 3*x* slower in full-system mode, confirming that our region-based DBT approach maintains its ability to optimise code across block boundaries, despite inserted interrupt checks.

Furthermore, an advantage of using the VirtIO infrastructure is that we can configure QEMU to use exactly the same kernel image, filesystem and block device configuration, allowing us to directly compare our I/O throughput against QEMU.

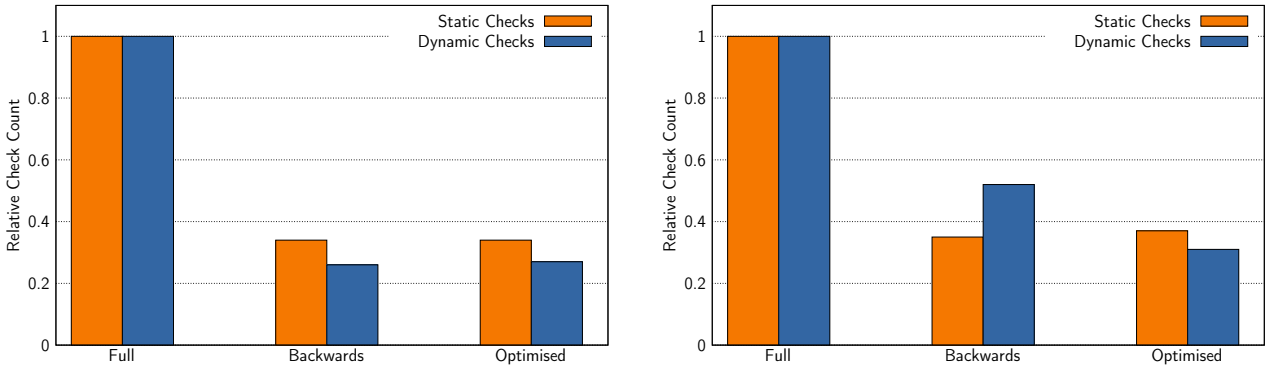
### 4.5 Further Analysis

#### 4.5.1 Static and Dynamic Interrupt Checks

A *static* interrupt check corresponds to the decision to place an interrupt check in a given basic block, where a *dynamic* interrupt check is an interrupt check that actually takes place at runtime. We are looking to minimise the number of *static* interrupt checks placed, and correspondingly reduce the number of *dynamic* checks made. A reduction in *static* interrupt checks serves two purposes:

a. The amount of IR the LLVM JIT compiler is presented with is lower, thereby reducing the amount of work the optimiser and compiler have to do and subsequently improving compilation time, and

b. the optimiser is free to perform more aggressive optimisations across the region, and produce better native code.




(a) I/O bound workload

(b) CPU bound workload

Figure 9: Reduction in *static* and *dynamic* interrupt checks for I/O and CPU-bound workloads on our three different interrupt checking schemes – lower is better. For I/O-bound workloads, our optimised placement algorithm reduces the amount of *static* checks by 66% and *dynamic* checks by 73%. For CPU-bound workloads, we reduce *static* checks by 63% and *dynamic* checks by 69%.

Figures 9a and 9b both show that we place fewer *static* interrupt checks, and as a consequence generally perform fewer *dynamic* checks. The exception to this is when using the *backwards branch* scheme in a CPU bound workload where we observe an increase in *dynamic* checks. This can be attributed to CPU-bound workloads spending more time in *hot* looping control-flow, where we will necessarily have inserted an interrupt check, and therefore increase the *dynamic* interrupt check count. Our optimised placement scheme places **66%** less interrupt checks than the baseline scheme, and causes **73%** fewer dynamic checks to occur.

#### 4.5.2 Interrupt Latency

The interrupt latency we are measuring is the time it takes for a simulated interrupt to be raised, until the time our execution engine begins executing the ISR. A reduction in interrupt latency will improve the throughput of an I/O bound workload, as data requests can be served more quickly. We measure the impact that our placement schemes have on interrupt latency, to ensure we are not deferring interrupts for an unacceptable period of time. These measurements are taken for the I/O-bound workload, as interrupt latency will not affect the throughput of CPU-bound workloads.

Whilst it may seem that we should have a lower latency for the *full* placement scheme (i.e. more checks, means more opportunities to respond to an interrupt) the impact that the scheme has on generated code quality is such that we actually observe higher latencies (**108µs**, over **80µs** on average) when employing this. Figure 10 shows that we reduce interrupt latency in the schemes which reduce the amount of *static* checks inserted, and for our optimised algorithm we reduce latency by **26%**. The higher quality of native code that we generate allows us to execute faster, and accounts for the fact that we can serve interrupts more quickly.

#### 4.5.3 Distribution of Interrupt Latencies

Figure 11 shows how various latencies for serviced interrupts are distributed between execution with the *full* and *optimised* scheme. The cosine similarity of the latencies produced between these schemes is $\cos\theta = 0.999$, indicating that the differing schemes do not significantly vary in the latencies yielded by the system. The cumulative latency distribution shown in Figure 11 is in fact comparable to that of the ARM port of the popular Xen hypervisor running on actual hardware [28].

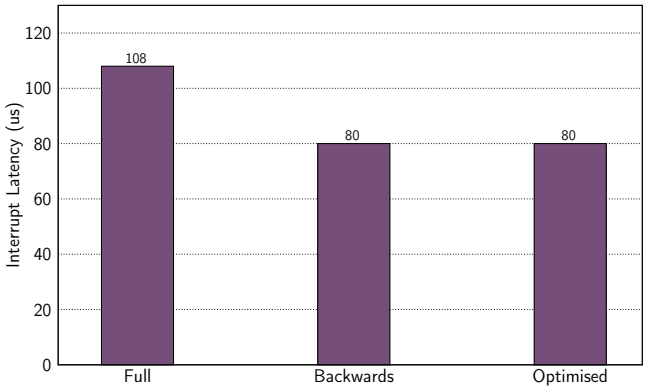



Figure 10: Absolute interrupt latency in $\mu s$ as measured when running the `hdparm` I/O benchmark – lower is better. We show that our optimised placement scheme reduces latency by 26%.

#### 4.5.4 Scalability

We have presented a range of interrupt frequencies, from 1Hz to 1kHz to test how our system scales to higher frequencies, and Figure 12 shows that our optimised scheme consistently performs better than the naïve full scheme. Furthermore, it can be seen that we maintain a relatively consistent level of performance across the frequency range, only dropping by approximately 2%.

#### 4.5.5 Comparison to Hardware

With simulation throughput approaching actual hardware performance, it's important to ensure that interrupt handling in our ISS is on even terms with the hardware we are emulating, i.e. we are not introducing an unacceptable amount of latency.

Interrupt response time observed on actual, non-simulated systems is the sum of a hardware dependent time and some operating system induced overhead. The hardware dependent time is determined by the micro-architecture of the processor and its current state, the system configuration and the type of interrupt. Operating system overheads may vary greatly between best and worst case scenarios, and are generally worst when the kernel (temporarily) disables interrupts.

According to the manufacturer's specification [2] the interrupt latency seen by a Linux driver running on an ARM1176JZ(F)-S with two levels of cache is approximately 5000 cycles. This is

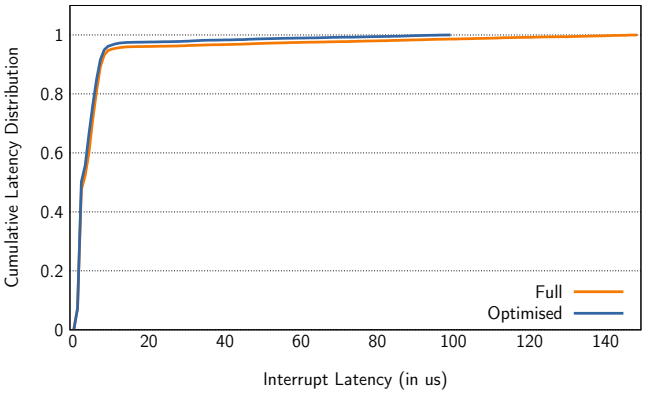



Figure 11: Cumulative distribution of interrupt latencies for our optimised interrupt policy, compared against checking every basic block.

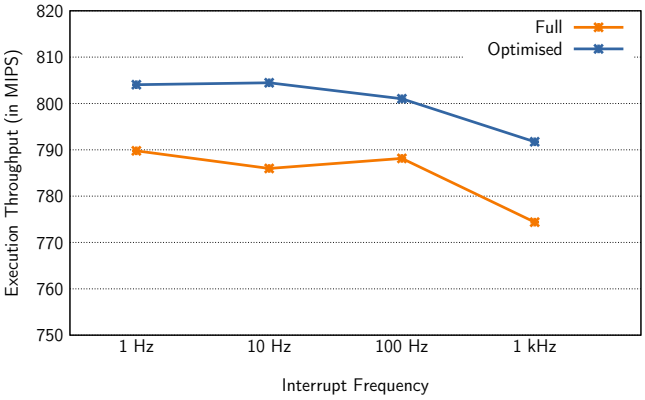



Figure 12: Comparison of full interrupt checking versus our interrupt checking policy using a range of interrupt frequencies from 1Hz to 1kHz. Full interrupt checking incurs a significant performance overhead at high interrupt frequencies, while our optimised policy continues to provide good performance.

largely caused by overheads in the operating system itself. 5000 cycles at 300MHz is 16.7µs, and our ISS yields an average latency of 80µs when configured with our optimised insertion algorithm.

## 5. Related Work

Sources of errors in full-system simulation have been recently analysed in [11]. The optimal placement of interrupt checks can be compared to the optimal insertion of profiling counters, a problem which can exhibit similar issues as described in the previous sections. However, updating profiling counters does not introduce additional control-flow – since the majority of cases are simple counter updates. Whilst reducing the number of counter updates can lead to performance improvements by reducing the amount of memory accesses, our problem is slightly different in that extra control-flow must be added to perform interrupt checks, thus causing additional latency in the optimiser, and resulting in less optimal code being generated. The technique described in [3] addresses the optimal placing problem, but does not address the issues that are encountered with additional exit points being introduced.

Whilst there are a number of full-system simulators available, either open-source (e.g. QEMU [4], ARM-Iss [18] or MARSSx86 [22]) or under a commercial license (e.g. Simics [19]), only few papers on interrupt handling in ISS have been published [7].

Older versions of QEMU utilised a zero-overhead interrupt checking scheme, which suffered from serious race-conditions, however later versions (including the version we compare against) have addressed these issues by inserting checks at the head of every basic-block. However the overall performance of our DBT is still on average 3.4*x* faster, due to techniques we employ based on those described in [6, 25, 27].

ARM-Iss [18] is an instruction set simulator for the ARM architecture. It is based on an interpretive execution model with additional instruction caching. Interpretive ISS are orders of magnitude slower than DBT ISS such as the one discussed in this paper. ARM-Iss checks for pending interrupts after each instruction. Whilst accurate this further exacerbates the performance penalty of this system.

MARSSx86 [22] is a full-system simulator for x86 CPUs. Under the hood, MARSSx86 uses QEMU for functional simulation and PTLsim for cycle-accurate modelling, using decomposition of x86 instructions into RISC-like $\mu$-ops and using basic block buffers to form traces of x86 $\mu$-ops. MARSSx86 delays the interrupt issued to the CPU until the CPU comes into the stable state, defined at opcode commit boundaries. Once the interrupts are issued to the CPU MARSSx86 switches from detailed simulation to functional emulation for correctly decoding the interrupt. The emulator mode sets up the correct CPU context to handle the interrupt but it does not start executing the interrupt handler. After the correct CPU context is set up, MARSSx86 switches back to the detailed simulation and starts simulating the interrupt handler code in kernel mode. Due to its cycle-accurate approach interrupt handling in MARSSx86 is precise, but it only operates at a speed of about 200 kilo instruction commits per second (KIPS), which is approximately 1000 times slower than the (instruction-accurate) DBT ISS presented in this paper.

An improved mechanism for the precise simulation of interrupts in cycle-accurate simulators has been presented in [7]. The simulator speculatively executes instructions of the emulated processor assuming that no interrupts will occur. At restore-points this assumption is verified and the processor state reverted to an earlier restore-point if an interrupt did actually occur. Whilst effective at speeding up cycle-accurate simulation this is still too costly for high-speed functional ISS.

A software simulator based on COTSon [1] that faithfully simulates x86 hardware at a speed in the tens of MIPS range has been described in [24]. Details on interrupt handling are not provided, though. Similarly, the strategies for interrupt checking are not further specified for Giano [10], SimFlex [12] or Graphite [21]. Gem5 [5] performs per-instruction interrupt checking due to its ambition to support cycle-accurate simulation.

### 5.1 Virtual Machines

Somewhat related to interrupt checking in an ISS is exception handling in a Java VM. Java exceptions are *synchronous*, though, i.e. they are related to the currently executed instruction and not triggered externally. Two techniques for dealing with Java exceptions during JIT compilation, namely *on-demand translation* of exception handlers and *exception handler prediction* are presented in [17]. In our system we implement a technique similar to *on-demand translation*, where translation of an interrupt handler is delayed until the interrupt really occurs. This, however, is a necessity of DBT in general and not specific to interrupt handling.

A notable exception is the implementation of *yield points* in the JikesRVM [14] Java VM, where interrupt checks are inserted in method prologues and epilogues, and on backedges. These checks are inserted to facilitate user-space scheduling of Java threads,

but have been deprecated (as of version 3.1.0) in favour of native threading. JikesRVM inserts a yield point in a method prologue and epilogue, and on a control-transfer instruction (such as an `if`) when the target is backwards. However, this technique is not applicable to our DBT, as we are discovering the structure of executing code dynamically, by building a control-flow graph of a region, and must take into account the possiblity of self-modifying code.

## 6. Summary & Conclusions

In this paper we have developed an optimised scheme for efficient placement of asynchronous interrupt checks in full-system instruction set simulators using region-based dynamic binary translation. Our technique detects control flow loops of any structure and nesting level and inserts a near-minimal number of interrupt checks. This technique provides correctness through the guarantee that at least one check for pending interrupts is performed for each iteration of any enclosing loop. On average, we reduce the number of dynamic interrupt checks in our ARM simulator by 73% in comparison to a scheme that checks for interrupts at the end of each basic block. Despite the reduced frequency of interrupt checks the latency for serving interrupts is reduced by 26% due to increased opportunities for code optimisation between interrupt checks. We also show that we maintain a performance advantage over state-of-the-art QEMU, where we improve I/O throughput by 1.6*x* and simulation performance by 3.4*x* in full-system simulation across a range of benchmarks.

### 6.1 Future Work

We intend to investigate the effect that the exact placement of interrupt checks has on the quality of generated code. For example, does the placement of an interrupt check enable the optimiser to produce better code when inserted into a loop condition block, as opposed to the loop body?

We will also investigate optimisation opportunities for synchronous exceptions/interrupts, particularly exceptions caused by memory instructions, which are frequently encountered.

## References


[1] Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., and Ortega, D. (2009). COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, **43**(1), 52–61.

[2] ARM Ltd. (2005–2009). ARM security technology building a secure system using TrustZone technology.

[3] Ball, T. and Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16**(4), 1319–1360.

[4] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX*, ATEC '05, pages 41–41, Berkeley, CA, USA. USENIX Association.

[5] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, **39**(2), 1–7.

[6] Böhm, I., Edler von Koch, T. J., Kyle, S. C., Franke, B., and Topham, N. (2011). Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA. ACM.

[7] Brandner, F. (2009). Precise simulation of interrupts using a rollback mechanism. In *Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '09, pages 71–80, New York, NY, USA. ACM.

[8] Charbit, P., Thomassé, S., and Yeo, A. (2007). The minimum feedback arc set problem is NP-hard for tournaments. *Comb. Probab. Comput.*, **16**(1), 1–4.

[9] Even, G., (Seffi) Naor, J., Schieber, B., and Sudan, M. (1998). Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, **20**(2), 151–174.

[10] Forin, A., Neekzad, B., and Lynch, N. L. (2006). Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, WA.

[11] Gutierrez, A., Pusdesris, J., Dreslinski, R., Mudge, T., Sudanthi, C., Emmons, C., Hayenga, M., and Paver., N. (2014). Sources of error in full-system simulation. In *Proceedings of 2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages pp. 13–22.

[12] Hardavellas, N., Somogyi, S., Wenisch, T. F., Wunderlich, R. E., Chen, S., Kim, J., Falsafi, B., Hoe, J. C., and Nowatzyk, A. G. (2004). SimFlex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, **31**(4), 31–34.

[13] Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, **34**(4), 1–17.

[14] Jikes RVM (2007). Threading and yieldpoints.

[15] Karp, R. M. (1972). *Reducibility among combinatorial problems*. Springer.

[16] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE.

[17] Lee, S., Yang, B.-S., and Moon, S.-M. (2004). Efficient Java exception handling in just-in-time compilation. *Softw. Pract. Exper.*, **34**(15), 1463–1480.

[18] Lv, M., Deng, Q., Guan, N., Xie, Y., and Yu, G. (2008). ARMISS: An instruction set simulator for the ARM architecture. In *International Conference on Embedded Software and Systems*, ICESS '08, pages 548–555.

[19] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, **35**(2), 50–58.

[20] Mark Lord (2012). hdparm(8): get/set sata/ide device parameters.

[21] Miller, J., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. (2010). Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12.

[22] Patel, A., Afram, F., Chen, S., and Ghose, K. (2011). MARSSx86: A Full System Simulator for x86 CPUs. In *Proceedings of the Design Automation Conference*, DAC '11.

[23] Russell, R. (2008). Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, **42**(5), 95–103.

[24] Ryckbosch, F., Polfliet, S., and Eeckhout, L. (2010). Fast, accurate, and validated full-system software simulation of x86 hardware. *IEEE Micro*, **30**(6), 46–56.

[25] Spink, T., Wagstaff, H., Franke, B., and Topham, N. (2014). Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pages 3–12. ACM.

[26] Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM Journal on Computing*.

[27] Wagstaff, H., Gould, M., Franke, B., and Topham, N. (2013). Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the Annual Design Automation Conference*, DAC '13, pages 21:1–21:6, New York, NY, USA. ACM.

[28] Yoo, S., Kwak, K.-H., Jo, J.-H., and Yoo, C. (2011). Toward under-millisecond I/O latency in Xen-ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 14:1–14:5, New York, NY, USA. ACM.